Adaptive Differential Filters for Fast and Communication-Efficient Federated Learning

Daniel Becking, Heiner Kirchhoffer, Gerhard Tech, Paul Haase,
Karsten Müller, Heiko Schwarz, and Wojciech Samek
Fraunhofer Heinrich Hertz Institute
Einsteinufer 37, 10587 Berlin, Germany
firstname.lastname@hhi.fraunhofer.de

Abstract

Federated learning (FL) scenarios inherently generate a large communication overhead by frequently transmitting neural network updates between clients and server. To minimize the communication cost, introducing sparsity in conjunction with differential updates is a commonly used technique. However, sparse model updates can slow down convergence speed or unintentionally skip certain update aspects, e.g. learned features, if error accumulation is not properly addressed. In this work, we propose a new scaling method operating at the granularity of convolutional filters which 1) compensates for highly sparse updates in FL processes, 2) adapts the local models to new data domains by enhancing some features in the filter space while diminishing others and 3) motivates extra sparsity in updates and thus achieves higher compression ratios, i.e. savings in the overall data transfer. Compared to non-scaled updates and previous work, experimental results on different computer vision tasks (Pascal VOC, CIFAR10, Chest X-Ray) and neural networks (ResNets, MobileNets, VGGs) in uni-, bidirectional and partial update FL settings show that the proposed method improves the performance of the central server model while converging faster and reducing the total amount of transmitted data by up to $377\times$.

1. Introduction

With current trends in "*Artificial Intelligence of Things*" and the rapidly growing number of intelligent devices [4], distributed computing on the edge becomes increasingly important. Federated learning (FL) allows multiple such edge devices to jointly train a deep learning model on their own local data and thus provides a basic level of privacy to the participating instances, since the local training data never leaves the devices. However, communication overhead is a major bottleneck hindering a straight-forward scalability of distributed learning systems. Client model updates or gradients are of the same size as the full model, which can be in the range of hundreds of megabytes for modern computer vision models [5]. To enable distributed training with many edge devices, extensive research on weight (update) compression has been conducted in recent years to improve computational and communication efficiency [25]. Communication efforts can be reduced through two basic concepts: 1) by reducing the communication frequency of weight updates, i.e. multiple local iterations of weight updates are performed before transmission; or 2) by compressing the data to be transmitted, e.g. by applying sparsification, quantization or encoding methods. Often, those methods slow down the training process, as they increase the number of iterations required to achieve a converged state of the neural network (NN) parameters.

Moreover, finding optimal communication protocol-specific hyperparameters (e.g., the number of communication rounds, the fraction of participating clients, or sparsification rates) becomes infeasible as the number of clients increases. Therefore, the authors of [2] suggest to develop easy-to-tune or auto-tuning algorithms for the instances that compose the federation, i.e., the client networks, instead of considering their network architectures, optimizers, and regularizations as already established and aiming only at the best accuracy of the central server model by optimizing the communication protocol. This is one of the motivations for our proposed filter scaling method, which is a minimally invasive intervention in the computational graphs of the neural networks, but has a significant impact on the training process. In this paper, we propose an FL pipeline in which lower communication frequency and compressed communication data do not compromise on convergence speed. With our studies we show that

- in various federated computer vision tasks, our proposed method quickly adapts the knowledge of clients' NNs from a given domain to new domains

- equipping NN layers with additional trainable scaling factors accelerates and regulates the entire FL process
- thanks to our sparsification, encoding and scaling technologies, the amount of transmitted data is minimal
- our proposed method leads to considerable speedups compared to prior work and improves communication efficiency.

First, we discuss the relation to previous works (Section 2), then we introduce our compression pipeline (Section 3), the filter scaling mechanism and the training protocol (Section 4). Section 5 presents experimental results, including the review of different optimization methods, the effects of filter scaling, computational overheads, variation in the number of clients and comparison with prior work. Section 6 concludes the paper.

2. Relation to Prior Work

For data reduction efficacy, it is crucial to use appropriate scaling factors in quantized representations of neural networks. That is, assuming an integer aligned uniform quantization scheme, the integer values are multiplied by a scalar step size to better capture the underlying data distribution. The work in [6] quantizes convolutional layers with one such step size per kernel and dense layers with a global step size. The step sizes are computed by a grid search, minimizing a mean squared error function. The authors showed that, given a small calibration data set, stochastic gradient descent (SGD) can further refine the step sizes for better approximation of the original model, improving top-1 accuracy by up to to 23%. The works in [3, 11] present direct step size learning in the scope of so-called *quantization-aware training* on a per layer granularity. As a counterpart, in non-uniform quantization each quantization level can be learned by a corresponding scaling factor independently, i.e. they are not equidistantly distributed. This technique finds application in ternary networks [18], i.e. having only two trainable quantization levels ("*centroids*") per network layer, or generally for low-precision networks to fine-tune centroids [9]. *Local Scaling Adaptation* is a technique that was adopted into the international standard ISO/IEC $15938-17$ on neural network compression (NNC) [8, 14] to increase the model capacity and thus compensate for quantization errors. It introduces a multiplicative factor for each output element of NN layers. These scaling factors are then applied to the quantized parameter representation before adding the bias.

In the context of distributed learning, several works compensate quantization errors by keeping track of the difference between the original and quantized gradients [24]. The work in [26] partitions gradients into blocks, where each gradient block (i.e. each tensor in their example) is compressed and transmitted in a 1-bit format with a corresponding scaling factor. Another method is to reduce the communication frequency and not send gradients each iteration but weight updates as generalized gradients after clients have performed multiple iterations, finally averaging the updates on the server side (*FedAvg*) [19]. The work in [20] demonstrates that also an *Adam* [13]-optimized variant of FedAvg converges faster than traditional SGD, however, at the cost of additionally transmitting per-parameter learning rates and estimates of the 1st and the 2nd raw moment of the gradient. Gradient sparsification methods only send gradient elements larger than a predefined threshold. One of the first approaches was presented by [23], using a predefined threshold, whereas in [16] a fixed sparsity rate is used. In practice, however, it is non-trivial to choose suitable thresholds or rates, as they can vary considerably for different neural architectures and even different layer types, layer locations within the neural network, or in different training iterations [1]. The architectures are predefined in most of the works, which may not be the optimal choice for the particular FL scenario. Beyond improving communication settings, *FedNAS* [10] proposes a paradigm for personalized FL by adapting client model architectures. To address the above challenges, we introduce statistics-dependent sparsification methods and additional trainable scaling factors to adapt the networks to sparse distributed training in divergent domains.

3. Compression Pipeline For Differential Neural Network Updates

In this work, one communication epoch t is defined as follows: 1) clients are synchronized with the server, i.e. they download the latest server update and add it to their model state, 2) clients optimize weights based on their local data, 3) clients compute the weight difference $\Delta\mathcal{W}$ wrt. the previous state:

$$\Delta\mathcal{W}_i^{(t)} = \mathcal{W}_i^{(t)} - \mathcal{W}_i^{(t-1)} \tag{1}$$

with $i \in I = [1, \ldots, \#\text{clients}]$, 4) compression of $\Delta\mathcal{W}_i^{(t)}$, 5) clients send the compressed $\Delta\hat{\mathcal{W}}_i^{(t)}$ to the server, 6) the server $\mathbb{S}$ applies federated averaging: $\Delta\mathcal{W}_{\mathbb{S}}^{(t)} = \frac{1}{|I|}\sum_{i\in I}\Delta\hat{\mathcal{W}}_i^{(t)}$, 7) the server sends the update $\Delta\mathcal{W}_{\mathbb{S}}^{(t)}$ to clients for synchronization, and so on. This basic communication protocol is in the spirit of federated averaging (FedAvg) [19]. Unless otherwise specified, no error accumulation is used.

One important part, that our proposed method also focuses on, is the compression of weight updates which is typically not part of FedAvg. In a first preprocessing step, sparsification techniques are applied, which set unimportant weight update elements to zero (i.e., weight update elements which minimally affect the model's computed output values; here we use the weight update magnitude as a

heuristic for importance), resulting in sparser and higher compressible weight update tensors.

Sparsification can be carried out in an unstructured or structured manner. In unstructured sparsification, any weight update element with small magnitude is set to zero, independently of its position. In contrast, in a structured sparsification an entire regular subset of parameters is set to zero, e.g., convolutional filters, kernels, matrix rows or columns. In our proposed method we use both paradigms, structured and unstructured sparsification. For unstructured sparsification we calculate the threshold θ_{u} parameter-wise by Gaussian approximation, i.e.

$$\begin{aligned}\theta_{\text{u}} = \max(&|\text{mean}(\Delta W) - \delta \cdot \text{std}(\Delta W)|,\\ &|\text{mean}(\Delta W) + \delta \cdot \text{std}(\Delta W)|)\\ &\text{s.t.} \quad \theta_{\text{u}} \geq \text{step\_size}/2\end{aligned} \tag{2}$$

Here ΔW corresponds to a specific layer/parameter update within the neural network update $\Delta\mathcal{W}$, std$(\cdot)$ describes the standard deviation of the parameter update distribution and δ is a hyperparameter which shifts the threshold and can be fine-tuned until a certain amount of sparsity or model performance degradation is exceeded. The step_size is a global parameter used for quantization. More precisely, in our uniform quantization scheme, it is an integer range multiplied by a float value which is used to generate quantization levels to which the original weight update distribution is assigned to, i.e. $[q, \ldots, -1, 0, 1, \ldots, p]\cdot$ step_size.

For structured sparsification, the regular subset of weight update elements we use is at the granularity of convolutional filters $F \in \mathbb{R}^{N\times K\times K}$ or output neurons $O \in \mathbb{R}^{N}$, when considering convolutional $W_{\text{conv}} \in \mathbb{R}^{M\times N\times K\times K}$ and dense layers $W_{\text{dense}} \in \mathbb{R}^{M\times N}$, respectively. Here, N refers to the number of input channels/elements, M to the number of output channels/elements and $K \times K$ to the convolutional kernel size. For simplicity, we use the notation of filter sparsification, which in the following shall also include sparsification of output neurons in dense layer types. As a threshold for structured sparsification, we calculate the average of filter means, parameter-wise:

$$\theta_{\text{s}} = \frac{\gamma}{M}\sum_{m=0}^{M-1}|\Delta\bar{F}|, \quad F \in \mathbb{R}^{N\times K\times K} \tag{3}$$

γ is again a hyperparameter which shifts the threshold and can be fine-tuned. Structured sparsity can be exploited very effectively in coding mechanisms, e.g. by skipping all-zero matrix rows that belong to the corresponding filters.

Having introduced unstructured and structured sparsity, the model update $\Delta\mathcal{W}$ is uniformly quantized and encoded with the NNC standard. Since the universal *DeepCABAC* entropy encoder of NNC can represent frequently occurring symbols with fewer bits, a high sparsity rate, i.e. a high probability that $\Delta w_{nkk} = 0$, reduces the model's entropy and thus results in smaller representations of $\Delta\mathcal{W}$.

4. Filter and Output Neuron Scaling for Distributed Learning Scenarios

In our proposed method, we introduce additional trainable scaling factors $\mathcal{S}$ to compensate sparse network updates $\Delta\mathcal{W}$. The trade-off is that the more zero elements are transmitted during a weight update, the less learning progress is possible because the current state of the model cannot change significantly. At the same time, we aim to have many zero elements to make the data more compressible. With our proposed scaling factors, we intend to balance this trade-off, i.e., we compensate for the fact that many update elements are zero by adjusting neighboring weights with a multiplicative factor, even if the associated neighborhood update is entirely zero.

The scaling factors are implemented at the granularity of convolutional filters/output neurons which ensures a low memory and computational overhead of the extra parameters. For the implementation, the computational graph of the neural network must be adapted by equipping convolutional and dense layers with a multiplication function and a multi-dimensional parameter whose first dimension's size corresponds to the number of output elements of the associated layer, while all following dimensions are of size 1 ("unsqueezed") to allow for correct tensor multiplication. The overall number of dimensions is equal to the associated layer's number of dimensions. E.g. for a convolutional layer $W_{\text{conv}} \in \mathbb{R}^{M\times N\times K\times K}$ the scaling factors are $S \in \mathbb{R}^{M\times 1\times 1\times 1}$, which allows each single filter $F \in \mathbb{R}^{N\times K\times K}$ in W_{conv} to be multiplied by a scalar s:

$$F^*_m = F_m \cdot s_m, \quad m \in [0, \ldots, M-1]. \tag{4}$$

In the following, s refers to a single scaling factor, i.e. $s \subseteq S \subseteq \mathcal{S}$. Apart from the increased model capacity, the additional parameters lead to different mechanisms taking effect in the context of incremental weight update compression of neural networks. Filter scaling speeds up the convergence of the center model, although very sparse weight updates typically require many communication rounds to converge. As shown in the experiments section, the scaling factors can not only amplify certain convolutional filters but also attenuate or even suppress redundant filters, i.e. feature extractors, and increase the overall sparsity of model updates $\Delta\mathcal{W}$.

Algorithm 1 gives an overview of our proposed filter-scaled sparse federated training. First, clients $\mathbb{C}_i$ are synchronized with the server $\mathbb{S}$ by receiving its averaged update, then each client is trained on its local data split $\mathcal{D}_i$. Note that in this training step the scaling factors $\mathcal{S}_i$ re-

Algorithm 1: Filter-scaled sparse federated training (unidirectional case)

input initial model $\mathcal{W}^{(0)}$, dataset $\mathcal{D}$, #clients, T, E
output federated learned server model $\hat{\mathcal{W}}_{\mathbb{S}}^{(T)}$
init all clients $\mathbb{C}_i, i \in I = [1, \ldots, \#\text{clients}]$ are initialized with the server $\mathbb{S}$ parameters $\mathcal{W}_i \leftarrow \mathcal{W}_{\mathbb{S}}$.
Every Client holds a different data split $\mathbb{C}_i \leftarrow \mathcal{D}_i, \mathcal{D}_{\text{val}_i}$.
Scaling factor initialization is $\mathcal{S}_i \leftarrow 1, \mathcal{S}_i \subseteq \mathcal{W}_i$.
for $t = 1, \ldots, T$ **do**
  Client side training:
  for $i \in I$ **do**
    download$_{\mathbb{C}_i \leftarrow \mathbb{S}}(\Delta \mathcal{W}_{\mathbb{S}}^{(t)})$
    $\mathcal{W}_i^{(t)} = \mathcal{W}_i^{(t)} + \Delta \mathcal{W}_{\mathbb{S}}^{(t)}$
    $\mathcal{W}_i^{(t+1)} = \text{train}(\mathcal{W}_i^{(t)}, \mathcal{D}_i)$, s.t. $\mathcal{S}_i^{(t+1)} = \mathcal{S}_i^{(t)}$
    $\Delta^s \mathcal{W}_i^{(t+1)} = \text{sparsify}(\mathcal{W}_i^{(t+1)} - \mathcal{W}_i^{(t)}, \mathcal{D}_i)$
    $\hat{\mathcal{W}}_i^{(t+1)} = \mathcal{W}_i^{(t)} + \Delta^s \mathcal{W}_i^{(t+1)}$
    $\mathcal{S}_i^{(1)} \leftarrow \hat{\mathcal{W}}_i^{(t+1)}$, perf $= \text{eval}(\hat{\mathcal{W}}_i^{(t+1)}, \mathcal{D}_{\text{val}_i})$
    for $e = 1, \ldots, E$ **do**
      $\mathcal{S}_i^{(e+1)} = \text{train}(\mathcal{S}_i^{(e)}, \mathcal{D}_i)$
      if $\text{eval}(\mathcal{S}_i^{(e+1)}, \hat{\mathcal{W}}_i^{(t+1)}, \mathcal{D}_{\text{val}_i}) \geq$ perf **then**
        perf = $\text{eval}(\mathcal{S}_i^{(e+1)}, \hat{\mathcal{W}}_i^{(t+1)}, \mathcal{D}_{\text{val}_i})$
        $\hat{\mathcal{W}}_i^{(t+1)} \leftarrow \mathcal{S}_i^{(e+1)}$
      end
    end
    $\Delta \hat{\mathcal{W}}_i^{(t+1)} = \hat{\mathcal{W}}_i^{(t+1)} - \mathcal{W}_i^{(t)}$
    upload$_{\mathbb{C}_i \rightarrow \mathbb{S}}(\Delta \hat{\mathcal{W}}_i^{(t+1)})$
  end
  Server side aggregation:
  $\Delta \mathcal{W}_{\mathbb{S}}^{(t+1)} = \frac{1}{|I|} \sum_{i \in I} \Delta \hat{\mathcal{W}}_i^{(t+1)}$
  $\hat{\mathcal{W}}_{\mathbb{S}}^{(t+1)} = \mathcal{W}_{\mathbb{S}}^{(t)} + \Delta \mathcal{W}_{\mathbb{S}}^{(t+1)}$
end
return $\hat{\mathcal{W}}_{\mathbb{S}}^{(T)}$

main unchanged. After optimizing the clients, the differences wrt. the previous state are calculated and sparsified according to Equations (2) and (3). The sparsified updates $\Delta^s \mathcal{W}_i^{(t+1)}$ are then added to the previous model state $\mathcal{W}_i^{(t)}$ which will serve as a basis for scaling factor optimization. For E sub-epochs, $\mathcal{S}_i$ are trained, whereas the rest of the network is frozen, including the running variances and means of the BatchNorm modules. After each sub-epoch, the scaled network is evaluated on the validation data set of the respective instance $\mathcal{D}_{\text{val}_i}$. From all sub-epochs, the network variant with the best validation performance is selected and proceeded with.

If the scaling factor training does not improve model performance compared to the sparsely updated model $\hat{\mathcal{W}}_i^{(t+1)}$, the scaling factor updates are discarded. Otherwise, the parameter differences of the model are recalculated (considering also the scaling factor parameters), quantized and encoded prior to transmitting them to the server. Note that we also investigated up- *and* downstream compression, where the server update $\Delta \mathcal{W}_{\mathbb{S}}^{(t+1)}$ is sparsified and scaled as well. This is not shown in Algorithm 1.

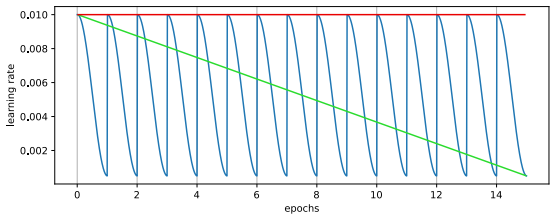

Figure 1. Learning rate schedules used with $|T|$ =15 epochs.

4.1. Learning Rate Scheduling for Scaling Factors

In our studies, we tested different optimizer-learning-rate-scheduler combinations, since the learning rate is crucial for appropriate convergence of the defined optimization problem. Even though the Adam optimizer [13] supports individual adaptive learning rate estimates for different parameters, the resulting learning rates are sometimes unsuitable for gradient optimization of our present FL scenarios. If the initial rate is too small, the optimizer is likely to converge to local minima and does not improve performance while simultaneously increasing the bitstream sizes due to the additional scaling factors. On the other hand, larger rates lead to coarser steps in gradient descent and thus are suboptimal for loss optimization in the later epochs. To limit Adam's peak learning rate, a linearly decreasing learning rate schedule can improve the results among others (e.g. the cosine annealing learning rate scheduler with warm restarts (CAWR) as proposed in [17]). The schedulers are implemented such that after each batch of inferenced data, the scheduler performs one step according to the functions shown in Figure 1. The warm restarts of CAWR are introduced after each main training epoch t, prior to training the scaling factors $\mathcal{S}$. We also ran experiments using SGD for scaling parameter optimization.

5. Experiments

In the experiments, we evaluate our novel compression pipeline for FL scenarios by deploying three widely used neural network families (MobileNet, ResNet, VGG) and three datasets which are introduced in this section. We investigated different optimization methods, communication protocols, scaling factor parameterizations and their effects on weight updates as well as computational overhead, variation in number of clients and comparison with prior work.

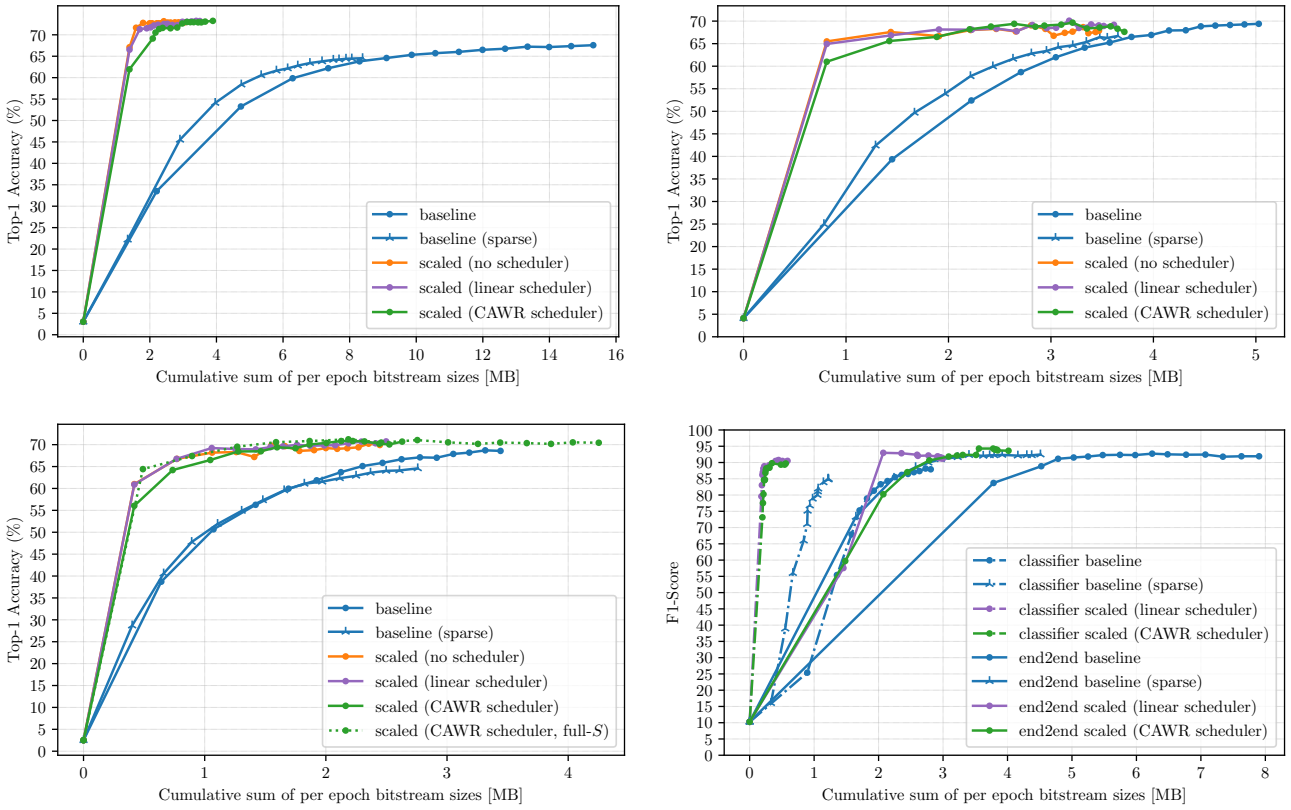

Figure 2. Filter-scaled sparse federated training of VGG11, ResNet18, MobileNetV2 and VGG16 (row-major order) solving the Pascal VOC and Chest X-Ray classification tasks, respectively. The MobileNetV2 chart includes experiments with scaling factors only attached to the output convolutions of each inverted residual block and one experiment with full scaling factors "full-S" for comparison. VGG16 results include bidirectional compression of partial and end-to-end updates.

5.1. Experimental Setup

The used neural networks are pre-trained on ImageNet and downloaded from the torchvision model zoo[1]. We adapted the classifiers to predict 20, 10 and 2 instead of $1,000$ classes for the computer vision datasets *Pascal VOC* [7], *CIFAR10* [15] and *Chest X-Ray* [12], respectively. Training and validation data was randomly split into non-overlapping client data sets $\mathcal{D}_i$. Note that for the sake of generalization, the optimization of scaling factors is evaluated locally using the validation data splits $\mathcal{D}_{\text{val}_i}$ and finally tested at the server's global model on the test data split. The batch sizes used are 64 for VOC and CIFAR and 100 for Chest X-Ray. The VGG11 network to solve the CIFAR task, named $\text{VGG11}_{\text{CIFAR10}}$ in the following, was thinned to $[32, 64, 128, 128, 128, 128, 128, 128]$ convolutional filters and 128 input neurons in the dense layers.

Unless otherwise specified, model weights $\mathcal{W}$ are transfer-learned in the FL scenarios as described in Sections 3-4 for $|T| = 15$ epochs using Adam with an initial learning rate of $1e-5$. For uniform quantization of weight updates $\Delta\mathcal{W}$, we use a step_size of $4.88e-4$ and $2.44e-4$ for uni- and bidirectional FL settings, respectively. Scaling parameter, bias and BatchNorm parameter updates are quantized with a step_size of $2.38e-6$. For further details on software implementation and data splits, we refer to the Appendix A.

[1] https://pytorch.org/vision/stable/models.html

5.2. Results for Different Optimization Schedules

In Figure 2 we show the federated training process in terms of server model performance and overall transmitted data between clients and server. Specifically, each data point in the charts represents one round of communication and indicates 1) how much data in bytes has overall (accumulative) been transmitted since the last round and 2) which top-1 accuracy (or which F1 score in the Chest X-Ray case due to large data imbalance) has been achieved by the aggregated server model. Thus, our goal is to move the curves as far as possible to the upper left corner of the charts, to achieve fast and communication-efficient FL.

We compare different configurations: The baseline con-

figuration neither includes filter scaling nor sparsification, the sparse baseline includes sparsification only and all other curves represent our proposed method of scaled and sparse differential filters with different optimizers (Adam, SGD) and learning rate schedulers (no schedule, linear, CAWR). It is evident that training converges significantly faster when applying filter scaling. Also, the cumulative sum of transmitted data is reduced, which leads to significant savings in data traffic, especially for VGG11. The Adam optimization of scaling factors outperforms SGD optimization in all cases, so we refer to the Appendix B for SGD results. For Adam optimization, linear and CAWR schedules improve the training process in different regimes: in later epochs, using CAWR generates models with higher performance, however, in the earlier epochs a linear schedule achieves better top-1 accuracy.

In literature, it has been shown that it is meaningful to compress weight updates in a bidirectional fashion, i.e. center-to-clients and clients-to-center communication. We tested this scenario with the Chest X-Ray dataset as it is motivated by a possible real world scenario, where a number of hospitals jointly train the detection of pathological evidence in x-ray data and a central server regularly updates the local detectors in the hospitals. As can be seen in Figure 2 (bottom right), also in the bidirectional compression scenario filter scaling can contribute to faster convergence, bitrate savings and improved model performance. And finally, we also compare a full model update ("end2end") with a partial update which only updates the classifier part of the VGG16 network consisting of a BatchNorm module and two dense layers. Here, only 258 scaling factors were applied, still the improvements are substantial.

The achieved gains seem to be counter-inuitive at first glance, as additional scaling parameters (with a fine quantization level, i.e. less lossy) have been added to the encoded bit stream and still the total bit rate has decreased. Thus, this characteristic is closer examined in the following section.

5.3. Effect of Filter-Tuning on Weight Updates

The scaling factors can amplify or suppress the impact of specific filters on the overall loss function. E.g., if a scaling factor s is close to zero, the entire convolutional filter only marginally contributes to the net output as the generated feature map will be quite sparse or of very low magnitude. This also has a large impact on the weight updates $\Delta\mathcal{W}$ and can be particularly advantageous for FL in computer vision applications to eliminate redundant or unused feature extractors learned in divergent image domains.

Having a closer look at Figure 3, we can observe different behaviors of scaling factors $\mathcal{S}$ dependent on their location within the neural network. On one hand, scaling factors in more shallow layers (close to the input) tend to converge to values close to 1, i.e. they do not change much and

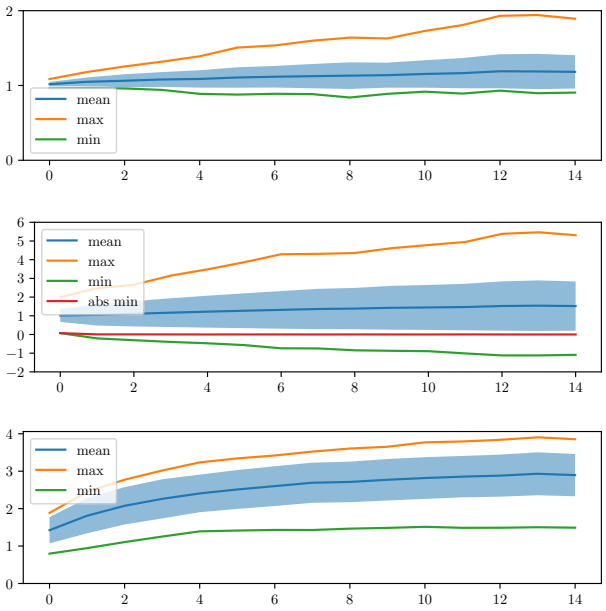

Figure 3. Scaling factor statistics for MobileNetV2 during 15 epochs of training of three layers at different network positions: first inverted residual block, 17th inverted residual block, and the dense output layer (from top to bottom).

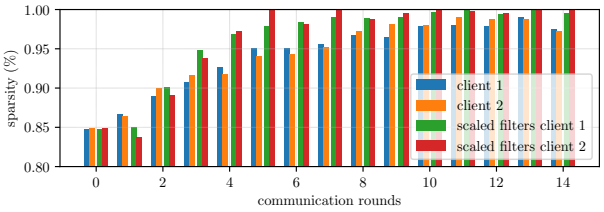

Figure 4. MobileNetV2 sparsity of two scaled clients vs. unscaled

thus only marginally influence the computational graph. In deeper located layers, the scaling factors on the other hand converge to values of $s \to 6$ while simultaneously suppressing other filters with values close to zero. Interestingly, in the dense output layer (bottom subplot of Figure 3) all output neurons are somehow amplified due to scaling.

A positive side-effect of these mechanisms is that also the filter weight updates $\Delta\mathcal{W}$ result in sparser tensors when using trainable scaling factors $\mathcal{S}$. It is not surprising that filters F with corresponding $s \to 0$ generate sparse output features. Consequently, the backpropagation algorithm computes smaller gradients for the associated weight updates. Another effect might be that large s make further training of filter weights redundant as the loss wrt. the filter F has already converged due to "macro training" through s. Figure 4 illustrates the sparsity of two clients per epoch when filter scaling and sparsification are applied vs. spar-

Table 1. Number of additional parameters and training time

model	$\#\text{params}_{\text{orig}}$	$\#\text{params}_{\text{add}}$	t_{add}
MobileNetV2	2.3M	2,836	1.17×
with full-$\mathcal{S}$	2.3M	17,076	1.31×
ResNet18	11.2M	4,820	1.62×
VGG11	128.8M	10,964	1.65×
$\text{VGG11}_{\text{CIFAR10}}$	0.8M	1,002	1.68×
VGG16	16.8M	4,482	1.60×
$\text{VGG16}_{\text{partial}}$	2.1M	258	1.40×

sification only. In the first few epochs, $\Delta\mathcal{W}$ is equal to or sparser than $\Delta\mathcal{W}_{\text{scaled}}$, however in most of the epochs filter scaling increases sparsity, even up to 100% (meaning that *exclusively* scaling factors $\mathcal{S}$ are sent by the client, i.e. macro training only). In summary, even though additional scaling factors are added to the transmitted bit streams, their impact increases the sparsity of filter updates, resulting in a final overall data reduction.

5.4. Computational and Memory Overhead

In addition to the experimental results above, Table 1 gives an insight into the number of scaling parameters $\mathcal{S}$ and additional training time required. It shows that $\mathcal{S}$ only accounts for 0.009% to 0.748% of the total network parameters, i.e. the extra storage and size of the compressed update is small. The additional processing time of scaling parameter training does not directly scale with the total number of $\mathcal{S}$. It also depends on the number of layers equipped with $\mathcal{S}$ and the overall size of the network. This is due to the fact that the optimization algorithm considers the whole computational graph of the network for gradient propagation, even if only $\mathcal{S}$ is updated. However, performing one training iteration to update $\mathcal{W}$ compared to performing two iterations, one for $\mathcal{W}$ and one for $\mathcal{S}$, requires on average only 1.17× to 1.68× the original computation time, which was also reflected in our experiments where we trained $\mathcal{S}$ for 5 sub-epochs E (i.e., training the scaling factors did not require 5× longer runtimes, but only ≈ 3× longer at most). This extra effort can be interpreted as an upper-bound and there are options to minimize the effort, e.g. by

1) applying scaling parameter training less frequently;

2) equipping less layers with $\mathcal{S}$, e.g. as tested with MobileNetV2, where we only equipped the final convolutional layer of each inverted residual block instead of all convolutional layers therein (cf. "full-$\mathcal{S}$" in Figure 2, bottom left and Table 1);

3) focusing $\mathcal{S}$ to be applied in deeper layers, as shown in partial updates of the VGG16 network, which converged as the end-to-end training counterpart (see also Section 5.3, where we showed that scaling factors do not change much in shallower layers);

4) considering smaller training splits to train $\mathcal{S}$, i.e. not the full training split which is used to train $\mathcal{W}$.

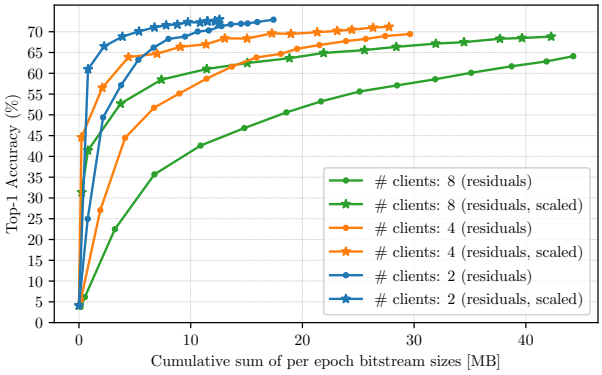

Figure 5. ResNet18 with error accumulation (residuals) and variation in number of clients.

5.5. Increasing the Number of Clients

The purpose of the following analysis is to investigate the potential scalability of our approach, i.e., can filter scaling still achieve fast and communication-efficient FL as the number of clients increases. In the Pascal VOC scenario, we increased the number of clients from 2 to 4 and 8. As this FL scenario with $|I| \geq 4$ barely achieved more than 30% top-1 accuracy *without* our filter scaling method, we implemented error accumulation as used in [21] after sparsification:

$$\Delta\mathcal{W}_i^{(t+1)} = \mathcal{R}_i^{(t)} + \mathcal{W}_i^{(t+1)} - \mathcal{W}_i^{(t)} \tag{5}$$

with $\mathcal{R}_i^{(t+1)} = \Delta\hat{\mathcal{W}}_i^{(t+1)} - \Delta\mathcal{W}_i^{(t+1)}$ and Equation 5 being inserted in Algorithm 1, line 10. The described error accumulation stores the difference of the compressed update and the original full-precision update locally ("residual"). According to this scheme, also small update elements in terms of magnitude can sum up until they exceed a certain threshold. Compared to the experimental results without residuals (cf. Figure 2, top right, vs. Figure 5, #clients: 2) residuals produce updates with a higher bitrate, as more information is accessible and will be send. The performance of the center model is slightly increased, however if the number of clients increases, convergence speed decreases as a consequence of the higher degree of distribution in the system and the rising non-IID-ness in data due to random partitioning of client train data (see Appendix C). As our primary goal is faster convergence and bitrate reduction, we don't address robustness on non-IID data in the scope of this work.

The results in Figure 5 show that filter scaling again outperforms all non-scaled training processes, which becomes even more evident as the number of clients increases. The relative improvement in top-1 accuracy is highest for the scenario with 8 clients equally involved in federated training, which may be a tentative indicator that the proposed method is scalable.

Table 2. Comparing different approaches from literature and combinations with parts of our proposed compression pipeline with our scaled and sparse filters method (Algorithm 1). Experiments were conducted using a thinned VGG11 solving CIFAR10 with 2, 4, 8 and 16 clients for $t = 1, ..., T = 90$ epochs. A constant sparsity rate of 96% was used for sparse ternary compression (STC) and our proposed methods.

	$\|I\| = 2$ Clients				$\|I\| = 4$ Clients				$\|I\| = 8$ Clients				$\|I\| = 16$ Clients			
	Acc. = 70.0		Acc. = **76.7**		Acc. = 64.3		Acc. = **71.6**		Acc. = 57.9		Acc. = **67.2**		Acc. = 48.2		Acc. = **61.4**	
	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t	$\sum$ data	t
FedAvg [19]	387.13 MB	58	∅	∅	747.09 MB	56	∅	∅	1,467.02 MB	55	∅	∅	2,988.37 MB	56	∅	∅
FedAvg [19]†	10.54 MB	58	∅	∅	17.82 MB	55	∅	∅	31.86 MB	55	∅	∅	55.48 MB	56	∅	∅
STC [21]†	4.33 MB	73	∅	∅	8.65 MB	74	∅	∅	16.56 MB	72	∅	∅	34.11 MB	76	∅	∅
Eqs. (2) + (3)	3.71 MB	90	∅	∅	7.23 MB	90	∅	∅	14.24 MB	90	∅	∅	27.46 MB	90	∅	∅
STC [21]‡	1.97 MB	31	5.53 MB	86	4.34 MB	34	10.59 MB	83	7.81 MB	31	21.79 MB	86	10.63 MB	22	42.81 MB	86
Algorithm 1	**1.68 MB**	36	**3.92 MB**	90	**3.61 MB**	39	**8.09 MB**	90	**6.16 MB**	34	**15.94 MB**	90	**7.93 MB**	23	**31.34 MB**	90

† Literature method with DeepCABAC encoding. ‡ Literature method with DeepCABAC encoding and our proposed filter scaling, cf. Equation (4).
∅ configuration has not achieved the target accuracy within 90 communication epochs

5.6. Comparative Results

After investigating several parameter variations of our proposed method, this subsection provides final results in comparison to current state-of-the-art methods. Here, Federated Averaging (FedAvg [19]) is a widely used baseline algorithm for communication-efficient FL. Accordingly, there are numerous works in the literature that propose improvements for FedAvg, of which Sparse Ternary Compression (STC [21]) is a regularly cited improvement [2, 22]. STC converges faster when compared to other averaging algorithms like FedAvg while less bits are communicated overall. Because of these advantages, we also compare our method with STC. The STC protocol compresses communication via sparsification, ternarization, error accumulation - according to Equation (5) - and Golomb encoding. For better comparability, and since DeepCABAC also makes use of Golomb encoding for binarization, we encoded all weight updates with DeepCABAC in our experiments.

Table 2 shows the respective experimental results. We deployed a thinned VGG11, as used in [21], which was federally learned on 80% of the CIFAR10 training data (20% were used as validation data). For the FL scenario, we used $2, 4, 8$ and 16 clients which were trained for $T = 90$ epochs with a constant sparsity rate of 96%.

First, we executed FedAvg, where the uncompressed client model updates are sent after each epoch, then averaged on the server side and broadcast to the clients (here without server-to-clients compression, which holds for all experimental results in Table 2). Second, we applied our proposed uniform quantization and DeepCABAC encoding to the FedAvg pipeline, which reduced the amount of communicated bits by a factor of ≈ 54. Third, STC is applied to the weight updates. Different from the experimental setup in [21], we did not use an "equivalent" delay period of n backpropagation iterations for FedAvg, consequently STC introduces additional sparsity, resulting in convergence in later epochs t. In the fourth row of Table 2, we provide results for applying our proposed sparsification to the weight updates which further decreases the amount of communicated bits, however, at the cost of additional training epochs to converge. This can be explained due to the structured sparsity introduced in the weight updates which is not fine-tuned here but fixed to 96% for better comparison. In a next step, we applied our filter scaling approach to STC and to our proposed sparsification scheme (i.e. "Algorithm 1").

With filter scaling enabled, both methods converge significantly faster, i.e. they require less communication rounds to achieve the target accuracies, while the amount of transmitted data is reduced by up to $\approx 377\times$ compared to FedAvg, and achieve higher top-1 accuracies overall compared to non-scaled configurations. In accordance with the previous section, the benefits of filter scaling increase as the population of clients grows. However, the accuracy achievable within 90 epochs, and thus the overall convergence speed, decreases with a larger number of participating clients (as also described in section 5.5).

6. Conclusion

In this paper we presented a fast converging compression pipeline for FL scenarios in computer vision applications. The pipeline applies structured and unstructured sparsification, and equips weight layers with additional trainable scaling factors at the granularity of convolutional filters. We showed that the scaling factors can amplify or suppress the impact of specific filters on the overall loss function and by doing so can compensate very sparse updates while improving convergence speed. As a result, overall data and bit stream sizes are reduced. The proposed method was tested and verified in its data reduction capability with different FL settings, including a learning scheduler variation, single (server-to-clients) and double (also clients-to-server) communication, end-to-end and partial updates, as well as FL systems with different number of clients. Compared to previous work, the proposed scaling method converges faster, achieves higher accuracy and reduces the amount of total transmitted data by up to $\approx 377\times$.

References

[1] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 440–445. Association for Computational Linguistics, 2017. 2

[2] Paolo Bellavista, Luca Foschini, and Alessio Mora. Decentralised learning in federated deployment environments: A system-level survey. *ACM Comput. Surv.*, 54(1), feb 2021. 1, 8

[3] Yash Bhalgat, Jinwon Lee, Markus Nagel, Tijmen Blankevoort, and Nojun Kwak. Lsq+: Improving low-bit quantization through learnable offsets and better initialization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020. 2

[4] Zhuoqing Chang, Shubo Liu, Xingxing Xiong, Zhaohui Cai, and Guoqing Tu. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet of Things Journal*, 8(18):13849–13875, 2021. 1

[5] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018. 1

[6] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018. IEEE, 2019. 2

[7] Mark Everingham, Luc Gool, Christopher K. Williams, John Winn, and Andrew Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. *Int. J. Comput. Vision*, 88(2):303–338, jun 2010. 5, 11

[8] Paul Haase, Daniel Becking, Heiner Kirchhoffer, Karsten Müller, Heiko Schwarz, Wojciech Samek, Detlev Marpe, and Thomas Wiegand. Encoder optimizations for the nnr standard on neural network compression. In *2021 IEEE International Conference on Image Processing (ICIP)*, pages 3522–3526. IEEE, 2021. 2

[9] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. 2

[10] Chaoyang He, Murali Annavaram, and Salman Avestimehr. Towards non-iid and invisible data with fednas: federated deep learning via neural architecture search. *arXiv preprint arXiv:2004.08546*, 2020. 2

[11] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019. 2

[12] Daniel Kermany, Kang Zhang, and Michael Goldbaum. Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification, Jan. 2018. Type: dataset. 5, 11

[13] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. 2, 4

[14] Heiner Kirchhoffer, Paul Haase, Wojciech Samek, Karsten Müller, Hamed Rezazadegan-Tavakoli, Francesco Cricri, Emre Aksu, Miska M Hannuksela, Wei Jiang, Wei Wang, et al. Overview of the neural network compression and representation (nnr) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 2021. 2

[15] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Apr. 2009. 5, 11

[16] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. 2

[17] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. 4

[18] Arturo Marban, Daniel Becking, Simon Wiedemann, and Wojciech Samek. Learning sparse & ternary neural networks with entropy-constrained trained ternarization (ec2t). In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020. 2

[19] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017. 2, 8

[20] Jed Mills, Jia Hu, and Geyong Min. Communication-efficient federated learning for wireless edge intelligence in iot. *IEEE Internet of Things Journal*, 7(7):5986–5994, 2019. 2

[21] Felix Sattler, Simon Wiedemann, Klaus-Robert Müller, and Wojciech Samek. Robust and communication-efficient federated learning from non-iid data. *IEEE transactions on neural networks and learning systems*, 31(9):3400–3413, 2019. 7, 8

[22] Osama Shahid, Seyedamin Pouriyeh, Reza M Parizi, Quan Z Sheng, Gautam Srivastava, and Liang Zhao. Communication efficiency in federated learning: Achievements and challenges. *arXiv preprint arXiv:2107.10996*, 2021. 8

[23] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015. 2

[24] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error compensated quantized sgd and its applications to large-scale distributed optimization. In *International Conference on Machine Learning*, pages 5325–5333. PMLR, 2018. 2

[25] Qi Xia, Winson Ye, Zeyi Tao, Jindi Wu, and Qun Li. A survey of federated learning for edge computing: Research problems and solutions. *High-Confidence Computing*, 1(1):100008, 2021. 1

[26] Shuai Zheng, Ziyue Huang, and James Kwok. Communication-efficient distributed blockwise momentum sgd with error-feedback. *Advances in Neural Information Processing Systems*, 32, 2019. 2

A. Further Information on Software Implementation and Datasets

In order to train scaling factors separately, an additional optimizer is instantiated, which only updates the scaling factors $\mathcal{S}$. All $s \in \mathcal{S}$ are initialized with the value 1 and then optimized with Adam or SGD with a momentum of 0.9. From a software perspective, we implemented a wrapper function, which detects all convolutional and dense layers within the respective neural network and replaces them with a scaled version of the respective module class. Note that this procedure changes the computational graph, however scaling factors have no effect on the original graph output if set to 1. All experiments were conducted on a homogeneous GPU cluster employing NVIDIA Ampere A100 GPUs (40 GB RAM). We use PyTorch 1.8.1 and torchvision 0.9.1 as deep learning framework and CUDA 11.1 for NN GPU acceleration.

The *Pascal Visual Object Classes Challenge 2012* [7] provides $11,540$ images categorized into 20 classes. The *Chest X-Ray* dataset [12] consists of $5,856$ images which are categorized into two classes: "pneumonia" and "normal". *CIFAR10* [15] consists of $60,000$ images with a resolution of 32×32 pixels, containing 10 classes. The data sets are split into approximately $60:20:20\%$, $75:15:10\%$ and $70:15:15\%$ for training:validation:testing purposes for Pascal VOC, Chest X-Ray and CIFAR10, respectively. We applied normalization and random horizontal flipping to all samples. Additionally, VOC samples are center cropped to 224×224 pixels and Chest X-Ray samples to 150×150 pixels.

B. SGD Results

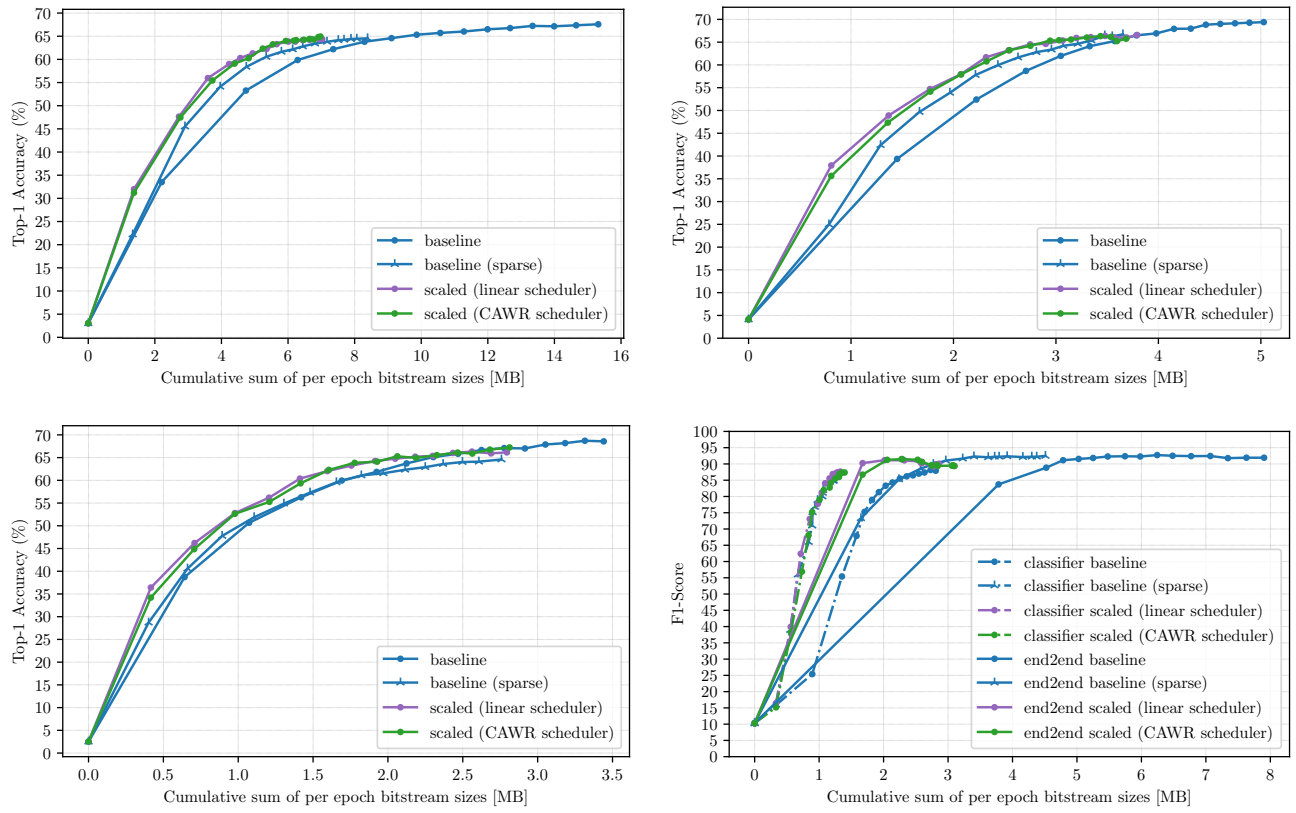

Figure B.1. SGD optimization for filter-scaled sparse federated training of VGG11, ResNet18, MobileNetV2 and VGG16 (row-major order) solving the Pascal VOC and chest x-ray classification tasks, respectively.

C. Data Distribution

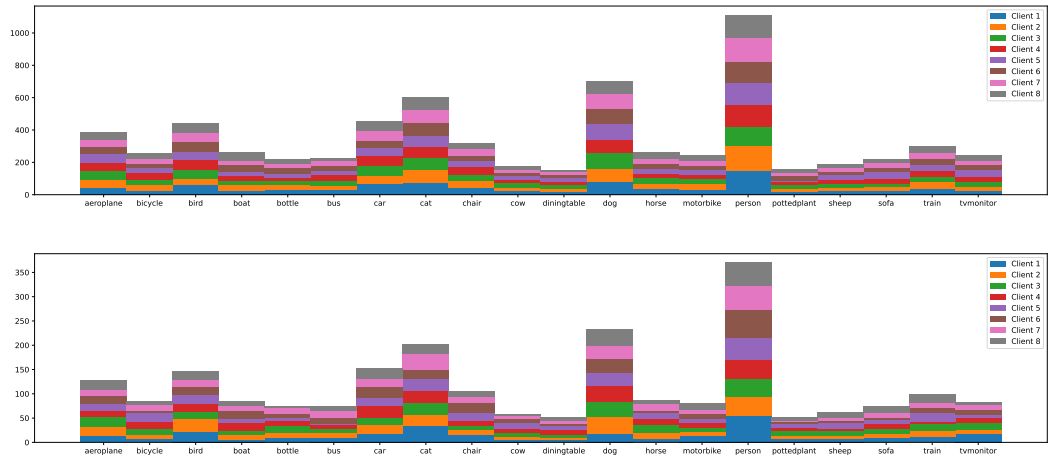

Figure C.1. Training (top) and validation (bottom) data distribution of the Pascal VOC scenario with 8 clients.

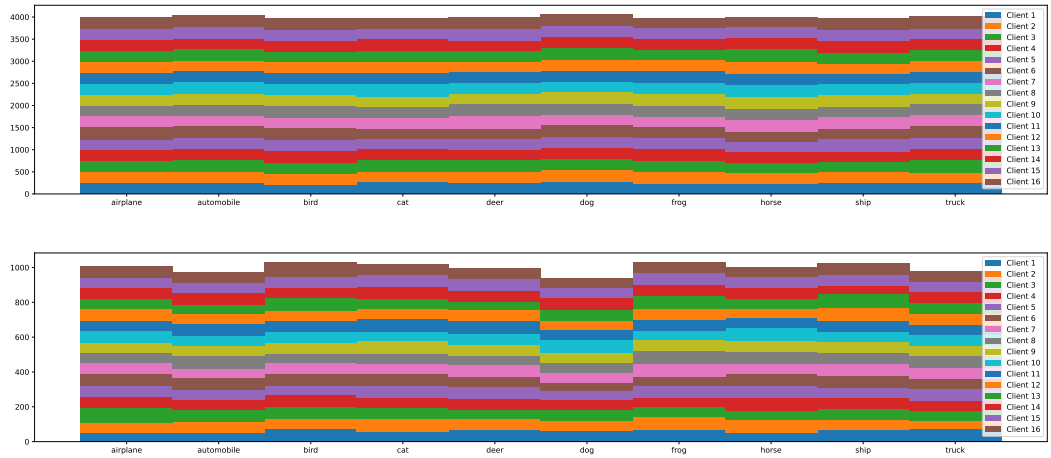

Figure C.2. Training (top) and validation (bottom) data distribution of the CIFAR10 scenario with 16 clients.